Patent Application of

Michael Stephen Fiske

for

EFFECTOR MACHINE COMPUTATION

I. FEDERALLY SPONSORED RESEARCH

NOT APPLICABLE.

II. COMPUTER PROGRAM IMPLEMENTING A STATIC EFFECTOR MACHINE

Attached with this patent application are two identical CD-ROMs. The following files are in directory *static_effector_machine*:

NAME	SIZE (bytes)	DATE OF CREATION
Machine.exe.txt	905,295	March 16, 2003
ExecutionParameters.xml.txt	2,834	March 17, 2003
MachineArchitecture.em.txt	1,402	March 16, 2003
MachineArchitectureSize.em.txt	73	December 16, 2002
MachineArchitectureTest.em.txt	1,045	March 16, 2003
MachineInputProgram.em.txt	59	March 16, 2003
MachineOutputProgram.em.txt	1,430	March 16, 2003
mainMachineExecution.cpp.txt	1,717	March 17, 2003

The file *machine.exe* is a software implementation of a static Effector machine. *machine.exe* is able to run properly in a Windows environment or an emulated Windows environment, assuming that the files, *ExecutionParameter.xml*, *MachineArchitecture.em*, *MachineArchitectureProgram.em*, *MachineArchitectureSize.em*, and *MachineInputProgram.em*, are in the same directory as *machine.exe*.

The file *mainMachineExecution.cpp* is source code that demonstrates how to program and execute the program on a static effector machine using the first method. In the function *main* of this file, the values of the machine architecture are read from the file *MachineArchitecture.em* and the static program is read from the file *MachineInputProgram.em*. The executable file *machine.exe* executes this static program on the machine architecture. After *machine.exe* executes, it stores the firing representation of the output effectors in the file *MachineOutputProgram.em*.

As proof of concept, the executable file *machine.exe* demonstrates the execution of a static Effector machine on a digital computer. However, the Effector machine realizes far greater computational power when its machine instructions are directly implemented in suitable hardware. A good hardware implementation should enable the Effector machine to execute computations more than four orders of magnitude faster than today's digital computers.

III. BACKGROUND – FIELD OF INVENTION

The Effector machine is a new kind of asynchronous computer. The Effector machine is not a digital computer. Its design has been influenced by Alan Turing's work in mathematics and computer science and Wilfred Rall's research in neurophysiology.

IV. ADVANTAGES OF INVENTION OVER PRIOR ART

In a standard digital computer, the only active computing elements are specialized registers in the microprocessor. Usually only one machine instruction can be computed at a time. This creates a computational bottleneck. The Effector machine overcomes this bottleneck because every computing element is active. When implemented in hardware, the Effector machine can execute multiple machine instructions simultaneously. This enormously increases the computing speed over current digital computers.

An Effector machine can be implemented in hardware built from circuits of transistors that operate subthreshold. This reduces the amount of power consumed by more than 5 orders of magnitude. Further, the amount of heat produced is greatly reduced, compared to a digital computer.

Executing a meta program whose instructions change the connections in a dynamic Effector machine, enables the Effector machine to perform tasks that digital computers are unable to compute.

V. DETAILED DESCRIPTION OF INVENTION

1. INTRODUCTION

An Effector machine consists of computing elements called effectors. There are three kinds of effectors: Input, Computational and Output effectors. Input effectors receive information from the environment, another Effector machine, or a program. Computational effectors receive messages from the input effectors and other computational effectors firing activity and transmit new messages to computational and output effectors. The output effectors receive messages from the input and computational effectors firing activity. The firing activity of the output effectors represents the output of the Effector machine. Every effector is an 'active element' in the sense that each one can receive and transmit messages simultaneously.

This paragraph describes the fundamental method of computation of a static Effector machine. Each effector receives messages, formally called pulses, from other effectors and itself and transmits messages to other effectors and itself. If the messages received by effector, E_i, at the same time sum to a value greater than the threshold,

then effector E_i fires. When an effector E_i fires, it sends messages to other effectors. There are three kinds of effectors, computational, output and input effectors.

2. MACHINE ARCHITECTURE

DEFINITION 2.1 *Static Machine*

The variables Γ, Λ, Δ represent index sets, used to index the input, computational, and output effectors, respectively. Depending on the particular machine architecture, it is possible for the intersections $\Gamma \cap \Lambda$ and $\Lambda \cap \Delta$ to be empty or non-empty. A *static machine* $\mathfrak{M}(\mathfrak{I}, \mathfrak{C}, \mathfrak{O})$ consists of a collection of input effectors, denoted as $\mathfrak{I} = \{E_i : i \in \Gamma\}$; a collection of computational effectors $\mathfrak{C} = \{E_i : i \in \Lambda\}$; and a collection of output effectors $\mathfrak{O} = \{E_i : i \in \Delta\}$. Each computational and output effector, E_i, has the following elements:

- A threshold θ_i.
- A refractory period r_i, where $r_i > 0$.
- A collection of pulse amplitudes, $\{A_{ki}\}_{k \in \Gamma \cup \Lambda}$.
- A collection of pulse widths, $\{\omega_{ki}\}_{k \in \Gamma \cup \Lambda}$, where $\omega_{ki} > 0$ for all $k \in \Gamma \cup \Lambda$.
- A collection of transmission times, $\{\tau_{ki}\}_{k \in \Gamma \cup \Lambda}$, where $\tau_{ki} > 0$ for all $k \in \Gamma \cup \Lambda$.
- A function of time, $\psi_i(t)$, representing the time effector i last fired.
  $\psi_i(t) = supremum\{s \in \mathbb{R} : s < t \;\; AND \;\; g_i(s) = 1\}$.
- A set of recent firing times of Effector k within Effector i's integrating window, $\mathfrak{F}_{ki}(t) = \{s \in \mathbb{R} : \textit{Effector } k \textit{ fired at time } s \textit{ and } 0 \le t - s - \tau_{ki} < \omega_{ki}\}$.
  Let $|\mathfrak{F}_{ki}(t)|$ denote the number of elements in the set $\mathfrak{F}_{ki}(t)$. If $\mathfrak{F}_{ki}(t)$ is the empty set, then $|\mathfrak{F}_{ki}(t)| = 0$.
- A collection of input functions, $\{\phi_{ki}(t)\}_{k \in \Gamma \cup \Lambda}$, each a function of time, and representing pulses coming from computational effectors, and input effectors.
  $\phi_{ki}(t) = |\mathfrak{F}_{ki}(t)| A_{ki}$.
- An output function, $g_i(t)$, representing whether the effector fires at time t.

$$g_i(t) = \begin{cases} 1 & if \quad \sum\limits_{k \in \Gamma \cup \Lambda} \phi_{ki}(t) > \theta_i \quad AND \quad (t - \psi_i(t)) \ge r_i. \\ 0 & otherwise. \end{cases}$$

An effector, E_i, can be an input effector and a computational effector. Likewise, an effector can be an output effector and a computational effector.

As defined above, when an output effector, E_i, is not a computational effector, $i \in \Delta \cap \tilde{\Lambda}$, E_i does not send pulses to effectors in this machine. This is captured

formally by the fact that the index k for the transmission times, pulse widths, pulse amplitudes, and input functions lies in $\Gamma \cup \Lambda$.

Input effectors that are not computational effectors have the same characteristics as computational effectors, except they have no input functions, $\phi_{ki}(t)$, coming from effectors in this machine. In other words, they don't receive pulses from effectors in this machine. Input effectors are assumed to be externally firable. An external source such as the environment or an output effector from another distinct machine $\mathfrak{M}'(\mathfrak{I}', \mathfrak{E}', \mathfrak{O}')$ can cause an input effector to fire. The input effector can fire at any time as long as this time minus the time the input effector last fired is greater than or equal to the input effector's refractory period.

Next, some vocabulary about the machine architecture is presented. If $g_i(t_0) = 1$, this means effector i fired at time t_0. The refractory period, r_i, is the amount of time that must elapse after effector E_i just fired before E_i can fire again. The transmission time, τ_{ki}, is the amount of time it takes for effector E_i to find out that effector E_k has fired. The pulse amplitude, A_{ki}, represents the strength of the pulse that effector k transmits to effector i after effector k has fired; after this pulse reaches E_i, the pulse width ω_{ki} represents how long the pulse lasts as input to effector E_i.

The expression *connection* from k to i represents the triplet $(A_{ki}, \omega_{ki}, \tau_{ki})$. If $A_{ki} = 0$, then one says that there is *no connection* from effector k to effector i; if $A_{ki} \neq 0$, then one says there is a *non-zero connection* from effector k to effector i.

DEFINITION 2.2 *Dynamic Machine*

A *dynamic machine* has the same definition as a static machine, except the *connections*, $(A_{ki}(s), \omega_{ki}(s), \tau_{ki}(s))$ change over time. Each of the connection variables is written as a function of s, which is a parameter for time. Thus, the pulse amplitudes, pulse widths and transmission times may change over time.

Stated in a more precise manner, replace the static machine definition of recent firing times with $\mathfrak{F}_{ki}(t) = \{s \in \mathbb{R} : \textit{Effector } k \textit{ fired at time } s \textit{ and } 0 \leq t - s - \tau_{ki}(s) < \omega_{ki}(s)\}$.

Furthermore, replace the static machine definition of input functions with the definition $\phi_{ki}(t) = \sum_{s \in \mathfrak{F}_{ki}(t)} A_{ki}(s)$, where $\{\phi_{ki}(t)\}_{k \in \Gamma \cup \Lambda}$ is a collection of input functions.

3. FIRING REPRESENTATIONS OF EFFECTORS

Consider effector E_i's firing times $F(E_i) = \{s : g_i(s) = 1\}$. If E_i's refractory period is greater than zero, arrange the elements of the set $F(E_i)$ into a sequence $[s_0, s_1, s_2, \dots]$, where $s_0 < s_1 < s_2 < \dots$.

DEFINITION 3.1 *Firing Representations*

Consider the interval of time $W = [t_1, t_2]$. Let s_m be the smallest element lying in W, and s_n the largest element lying in W. Then E_i's firing sequence within the *window of computation* W is $F(E_i, W) = [s_m, s_{m+1}, \ldots, s_n]$, where $s_m < s_{m+1} < \cdots < s_{n-1} < s_n$, and $\bigcup_{k=m}^{n} \{s_k\} = \{s \in W : g_i(s) = 1\}$. The sequence $F(E_i, W)$ is called a *firing representation* of the effector E_i over the interval of time W. With a collection of effectors $\{E_0, E_1, E_2, \ldots\}$, create the infinite tuple $(F(E_0, W), F(E_1, W), F(E_2, W), \ldots)$. The infinite tuple $(F(E_0, W), F(E_1, W), F(E_2, W), \ldots)$ is called a *firing representation* of the effectors $\{E_0, E_1, E_2, \ldots\}$ over the *window of computation* W.

At a fundamental level of interpretation, firing representations express the input to, the computation of, and the output of an Effector machine. At a more abstract level, firing representations can represent an input symbol, an output symbol, a sequence of symbols, a number, or even a sequence of program instructions.

EXAMPLE 3.2

For each real number x in $[0, 1]$, there exists a distinct firing representation of effectors $\{E_0, E_1, E_2, \ldots\}$ over a finite interval of time.

Proof: Let $\gamma > 0$ be a finite real number. Choose the interval of time for each effector to be $[0, \gamma]$. For any time $t < 0$, there are no restrictions on when effectors E_i fired. With this in mind, for each i choose the refractor period $r_i = \frac{\gamma}{2}$ so that it is possible for effector E_i to fire or to not fire during the interval of time $[0, \gamma]$. The sequence $(b_0, b_1, b_2, b_3 \ldots)$ is a binary representation of a real number x in $[0, 1]$. If $b_i = 1$, then choose E_i to fire at least once during the interval of time $[0, \gamma]$. If $b_i = 0$, then choose E_i to not fire during the interval of time $[0, \gamma]$. This shows the existence of a firing representation for x. To show distinctness, verify that if $x \neq y$, then the firing representation for x is distinct from the firing representation for y. Let $(b_0, b_1, b_2, b_3, \ldots)$ denote the binary representation for x and $(c_0, c_1, c_2, c_3, \ldots)$ denote the binary representation for y. Since $x \neq y$, then $c_k \neq b_k$ for some k. In the first case, $b_k = 0$ and $c_k = 1$. (The analysis for the other case $b_k = 1$ and $c_k = 0$ is apparent from symmetry.) By our method of construction, the E_k effector for x does not fire during the interval of time $[0, \gamma]$; and the E_k effector for y fires during the interval of time $[0, \gamma]$. Thus, the firing representations for x and y are distinct. □

DEFINITION 3.3 *Sequence of Firing Representations*

$\mathfrak{C} = \{E_0, E_1, E_2, \ldots\}$ denotes a collection of effectors. Let $W_1, W_2, W_3, \ldots, W_n$ be a sequence of time intervals. Let $F(\mathfrak{C}, W_1) = (F(E_0, W_1), F(E_1, W_1), F(E_2, W_1), \ldots)$ be a firing representation over the interval W_1. Let $F(\mathfrak{C}, W_2) = (F(E_0, W_2), F(E_1, W_2), F(E_2, W_2)\ , \ldots)$

be a firing representation of the effectors $\{E_0, E_1, E_2, \ldots\}$ over the interval of time W_2. In general, let $F(\mathfrak{E}, W_i) = (F(E_0, W_i), F(E_1, W_i), F(E_2, W_i), \ldots)$ be the firing representation over the interval of time W_i. From these one can create a *sequence of firing representations,* $F(\mathfrak{E}, W_1), F(\mathfrak{E}, W_2), F(\mathfrak{E}, W_3), \ldots, F(\mathfrak{E}, W_n)$.

4. MACHINE COMPUTATION

DEFINITION 4.1 *Machine Computation*

Let $F(\mathfrak{E}, W_1), F(\mathfrak{E}, W_2), F(\mathfrak{E}, W_3), \ldots, F(\mathfrak{E}, W_n)$ denote a sequence of firing representations. Let $F(\mathfrak{E}, S_1), F(\mathfrak{E}, S_2), F(\mathfrak{E}, S_3), \ldots, F(\mathfrak{E}, S_n)$ be a sequence of firing representations. If there exists a machine architecture for an Effector machine whose input effectors with a sequence of firing representations $F(\mathfrak{E}, S_1), F(\mathfrak{E}, S_2), F(\mathfrak{E}, S_3), \ldots, F(\mathfrak{E}, S_n)$, generate with its output effectors $F(\mathfrak{E}, W_1), F(\mathfrak{E}, W_2), F(\mathfrak{E}, W_3), \ldots, F(\mathfrak{E}, W_n)$, then one says that the machine computes $F(\mathfrak{E}, W_1), F(\mathfrak{E}, W_2), F(\mathfrak{E}, W_3), \ldots, F(\mathfrak{E}, W_n)$

DEFINITION 4.2

An Effector machine is an *interpretation between two sequences of firing representations* if the machine can compute the output sequence of firing representations from the input sequence of firing representations.

When using a dynamic machine, it is possible to use a distinct machine architecture for each distinct sequence of firing representations described in Definition 4.2.

DEFINITION 4.3 *Static Program*

$\mathfrak{M}(\mathfrak{I}, \mathfrak{E}, \mathfrak{O})$ denotes an Effector machine. $\mathfrak{M}$ can be a static or dynamic machine. A *static program* is a sequence of firing representations presented to $\mathfrak{M}(\mathfrak{I}, \mathfrak{E}, \mathfrak{O})$'s input effectors, $\mathfrak{I}$.

A meta-program determines how to change a dynamic machine's architecture as it executes.

DEFINITION 4.4 *Meta Program*

$\mathfrak{M}(\mathfrak{I}, \mathfrak{E}, \mathfrak{O})$ denotes a dynamic machine. For each j, the symbol x^j is the symbol A, ω or τ, representing a pulse amplitude, a pulse width or a transmission time, respectively. A *meta-program* is a finite sequence of quintuples $[(x^1, k_1, i_1, v_1, t_1), (x^2, k_2, i_2, v_2, t_2), \ldots, (x^n, k_n, i_n, v_n, t_n)]$ where each t_i represents a time and $t_1 < t_2 < \cdots < t_n$. For each j, where $1 \leq j \leq n$, the quintuple, $(x^j, k_j, i_j, v_j, t_j)$, instructs $\mathfrak{M}(\mathfrak{I}, \mathfrak{E}, \mathfrak{O})$ to assign the value v_j to connection element, ${x^j}_{k_j i_j}$, at time t_j. In particular, at time t_1, connection element, ${x^1}_{k_1 i_1}$, is assigned the value v_1. If x^1 is the symbol A, then pulse amplitude $A_{k_1 i_1}$ is assigned the value v_1 at time t_1. Similarly, at time t_2, connection element, ${x^2}_{k_2 i_2}$, is assigned the value v_2. If x^2 is the symbol ω, then pulse width $\omega_{k_2 i_2}$ is assigned the value v_2 at time t_2.

EXAMPLE OF PROGRAMMING AN EFFECTOR MACHINE

In the enclosed CD-rom, there is a file *MachineInputProgram.em* that contains an example of a Static program. The first line of this Static program has 0 in the time column and 0 in the effector column. The first line instructs that input effector 0 is fired at time 0. The second line has a 5 in the time column and a 0 in the effector column. This second line instructs that input effector 0 is fired at time 5. The last instruction is on the third line. The third line instructs that input effector 0 is fired at time 11. The simple static program in *MachineInputProgram.em* helps illustrate how a static program is interpreted by a static Effector machine. Longer more complex static programs can also be executed with *machine.exe.*

In the file *MachineOutputProgram.em*, the first line of this static program has a 3.00 in the time column and 0 in the effector column. The first line means that output effector 0 fired at time 3.00. The second line of this static program has a 4.00 in the time column and 0 in the effector column. The second line means that output effector 0 fired at time 4.00. The rest of the lines in file *MachineOutputProgram.em* are interpreted in a similar way.

The output file, *MachineOutputProgram.em*, has the same format as the input file, *MachineInputProgram.em*, so the output of one machine can be used as a static program for another machine. This enables one to compose the results of one or more Effector machines. Using machine composition, a programmer can program a complex task with many small Effector machines that perform simpler tasks.

5. INPUT AND OUTPUT INTERPRETER

Two different methods of programming an Effector machine are defined here. In the first method, called *explicit programming*, someone explicitly designs a machine architecture, and writes a Static program or Meta program.

The design of a machine architecture and a program, using *explicit programming*, were explained in sections 2, 3, and 4. Despite the fact that machine instructions for an Effector machine are quite different from a digital computer's machine instructions, *explicit programming* is analogous to writing a computer program in assembly language, C, or some other programming language for a digital computer.

A second method, called *implicit programming*, designs the architecture and programs of an Effector machine based on how you want the machine to behave. A method for *implicit programming* is presented in the next section.

In both explicit programming and implicit programming, the use of an *Input Interpreter* and *Output Interpreter* greatly simplify the programming task. In any programming

endeavor, the task that needs to be solved, performed or finished can be represented mathematically by an *(input, output)* set of pairs, $\{(I_1, O_1), (I_2, O_2), \ldots\}$. In some cases, the set, $\{(I_1, O_1), (I_2, O_2), \ldots (I_n, O_n)\}$, is finite. In other cases, it can be infinite. For each i, I_i is the input point, and O_i is the corresponding output point. The *(input, output)* set specifies the behavior of the Effector machine that we wish to design.

An input and output interpreter translates a chosen pair (I_i, O_i) to the native machine instructions of the Effector machine; specifically, the input interpreter translates the point I_i, into a sequence of firing activity executed by the input effectors, $\mathfrak{I}$. The output interpreter translates the firing activity of the output effectors, $\mathfrak{O}$, into an output point, O_i. This output point represents the output of the effector machine. On the attached CD-rom, the input interpreter and output interpreter are designed with C^{++} source code and can change, depending on what the Effector machine is designed to do. However, an input interpreter can be implemented as a distinct Effector machine. Likewise, an output interpreter can be implemented as a distinct Effector machine.

6. EVOLUTIONARY DESIGN OF AN EFFECTOR MACHINE

Implicit programming on an Effector machine can be accomplished using evolutionary methods. Evolutionary methods for extremely different applications were first introduced by [Box], [Bledsoe], [Bremermann], and [Friedman].

Implicit programming is preferable to explicit programming when the programmer knows what she wants the Effector machine to accomplish, but she doesn't know how to explicitly design the Effector machine architecture, Static program or Meta program. This is analogous to a CEO of a company asking her engineers to build a new computer, but she does not know how to build this computer. The CEO knows how she wants the computer to behave, but does not know how to explicitly build it.

This paragraph introduces some definitions needed to present an evolutionary design process, called *Cyclic Graph Evolution.* The number of Effector machines in each generation of the evolutionary process is m, where m is an even number. The symbol $p_{crossover}$ denotes the probability that two machines chosen for the next generation will be crossed over. The symbol $p_{mutation}$ denotes the probability that a machine will be mutated. The symbol θ represents an optimal fitness value. In other words, if an Effector machine has a fitness value greater than θ, then it can competently compute the *(input, output)* set of pairs, $\{(I_1, O_1), (I_2, O_2), \ldots\}$.

This paragraph presents the primary steps of *Cyclic Graph Evolution*, which will be abbreviated with the letters, CGE. CGE designs at least one effector machine that competently computes the collection of *(input, output)* pairs, $\{(I_1, O_1), (I_2, O_2), \ldots (I_n, O_n)\}$.

Build an initial generation of Effector machines, $A = \{\mathfrak{M}_1, \mathfrak{M}_2, \mathfrak{M}_3, \ldots, \mathfrak{M}_m\}$. Determine the fitness of each Effector machine, $\mathfrak{M}_j$. The fitness of $\mathfrak{M}_j$ is determined by its overall ability to represent, O_i, with its output effectors, after $\mathfrak{M}_j$'s input effectors receive input I_i. The fitness is also dependent on the amount of memory $\mathfrak{M}_j$ consumes and the average amount of computational time that it takes to compute the outputs O_i. A smaller amount of memory increases the fitness. A smaller computational time increases the fitness also. During one generation, CGE randomly chooses two machines from the current generation based on their fitness. Sometimes these two machines selected are crossed over and mutated. After a possible crossover and mutation, these two machines are placed in the next generation. The selection of two machines, and possible crossover and mutation is repeated until the next generation has m machines. CGE is repeated until the best Effector machine $\mathfrak{M}_{best}$ has a fitness greater than θ.

The CGE design process is presented below as an algorithm:

CYCLIC GRAPH EVOLUTION DESIGNS EFFECTOR MACHINES

Build an initial population of machines, $A = \{\mathfrak{M}_1, \mathfrak{M}_2, \mathfrak{M}_3, \ldots, \mathfrak{M}_m\}$.

while(true)

{

Set G equal to the empty set. (G stores the machines in the next generation.)

For each j in $\{1, 2, 3, \ldots, m\}$

{

For each i in $\{1, 2, 3, \ldots, n\}$, find $\mathfrak{M}_j$'s overall ability to represent, O_i, with its output effectors, after $\mathfrak{M}_j$'s input effectors receive input I_i.

Store this ability as the fitness of $\mathfrak{M}_j$.

}

Set q equal to the number of machines with fitness greater than θ.

if ($q \geq 1$) exit the loop *while(true)* and return $\mathfrak{M}_{best}$, the machine with the highest fitness.

while(the size of $G < m$)

{

Randomly choose two machines, $\mathfrak{M}_j$, $\mathfrak{M}_k$, from A for the next generation. (The probability of choosing $\mathfrak{M}_j$ is proportional to its fitness.)

Randomly choose a number r between 0 and 1.

If $r < p_{crossover}$, then crossover machines $\mathfrak{M}_j$ and $\mathfrak{M}_k$.

Randomly choose numbers s_1, s_2 between 0 and 1.

If $s_1 < p_{mutation}$, then mutate machine $\mathfrak{M}_j$.

If $s_2 < p_{mutation}$, then mutate machine $\mathfrak{M}_k$.

Set G equal to $G \cup \{\mathfrak{M}_j, \mathfrak{M}_k\}$.

}

Set A equal to G.

}

FURTHER DETAILS ON CYCLIC GRAPH EVOLUTION

A.) This part covers in further detail the construction of the effector machines in the initial population. Each machine is constructed separately. The stucture of how connections and effectors are organized in a machine must be explained first. Each machine that is designed by cyclic graph evolution is composed of one Input module, one Output module, and one or more Internal modules. (Refer to the diagram titled *Effector Machine Structure used in Cyclic Graph Evolution.*) Every effector lies in exactly one module. This structure does not affect how the Effector machine computes, as explained in Section 2, but it does affect how two machines are created for CGE, crossed over and mutated. A non-zero connection between two effectors lying in two distinct modules is called an *external connection.* A non-zero connection between two effectors lying in the same module is called an *internal connection.*

Please refer to the file *EvolutionConstants.xml*, stored on the CD-ROM, as we discuss the constants used in the construction of a machine. For each machine created, an Input Module and an Output Module are created. Further, a random number k is chosen such that **MIN_MODULES_PER_MACHINE** $\leq k \leq$ **MAX_MODULES_PER_MACHINE**, and k internal modules are created.

Each module is constructed separately. External connections are added afterwards. For each module, a random u is chosen such that **MIN_EFFECTORS_PER_MODULE** $\leq u \leq$ **MAX_EFFECTORS_PER_MODULE**.

Next, u effectors are created. For each effector, the refractory period is randomly chosen between **MIN_REFRACTORY_PERIOD** and **MAX_REFRACTORY_PERIOD**. For each effector, the threshold is randomly chosen between **MIN_THRESHOLD** and **MAX_THRESHOLD**. Next, for each effector, a random number v is chosen such that **MIN_NUM_CONNECTIONS** $\leq v \leq$ **MAX_NUM_CONNECTIONS**. This means that v connections are created for this particular effector.

For each of these v connections, the value of the pulse amplitude is randomly chosen between **MIN_AMPLITUDE** and **MAX_AMPLITUDE**, inclusive. For each connection, the value of the pulse width is randomly chosen between **MIN_PULSE_WIDTH** and **MAX_PULSE_WIDTH**, inclusive. For each connection, the transmission time is ran-

domly chosen between **MIN_CONDUCTION_TIME** and **MAX_CONDUCTION_TIME**, inclusive.

B.) This part explains how CGE executes a crossover between two machines, machine A and machine B. Please refer to the diagram titled, *CGE Crossover.* The variable n_1 represents the number of internal modules in the machine A, and n_2 represents the number of internal modules in machine B. For machine A, a random whole number j_1 is chosen, lying between **LOWER_FRACTION_NUM_MODULES** $*$ n_1 and **UPPER_FRACTION_NUM_MODULES** $* n_1$. For machine B, a random whole number j_2 is chosen, lying between **LOWER_FRACTION_NUM_MODULES** $* n_2$ and **UPPER_FRACTION_NUM_MODULES** $* n_2$.

In the diagram, for machine A, $j_1 = 2$ was selected. For machine B, $j_2 = 1$ was selected. Since $j_1 = 2$, two distinct numbers are chosen randomly from the set $\{1, 2, \ldots, n_1\}$. For machine A, these two numbers are 2 and 3. Since $j_2 = 1$, one number is randomly chosen from $\{1, 2, \ldots, n_2\}$. In this case, 2 was chosen. What all this means is that internal modules m_2 and m_3 of Machine A are crossed over with internal module m_2 of Machine B. All the external connections to these modules are also severed. In the diagram, after crossover, new external connections are created and added to the internal modules that were crossed over. Observe that internal connections in a module that is crossed over are not changed or severed after crossover. This is why these objects are called modules. Internal connections are not severed by a crossover.

C.) This part further discusses mutations of an Effector Machine. There are multiple ways to mutate an Effector machine. An effector can be added to a module along with some non-zero connections, connecting the new effector to effectors inside the same module, and possibly connecting it to effectors in different modules. A second type of mutation deletes an effector. In this case, all of the deleted effector's non-zero connections are removed also.

A third type of mutation may change the pulse amplitude, pulse width, or the transmission time of a connection. Similar to adding an effector, a fourth type of mutation adds a new connection. Similar to removing an effector, a fifth type of mutation removes a connection from the Effector machine.

A sixth type of mutation creates a new module containing new effectors and connections between them and connects this new module to other modules with new external connections. A seventh type of mutation deletes a module and removes all of the deleted module's external connections.

The file *EvolutionConstants.xml* contains constants that guide these seven different types of mutations. In addition, implementations of these mutations in C^{++} source

code are in the directory *evolution* on the enclosed CD-ROM. Generally, $p_{crossover}$ should usually range from 0.3 to 0.7. Also, $p_{mutation}$ should usually be less than 0.1.

D.) Previous evolutionary methods evolve directed graphs that contain no cycles. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a finite set of n vertices. An edge in a directed graph is an ordered pair (v_i, v_j) with each vertex lying in V. Let $E = \{(v_{i_1}, v_{k_1}), (v_{i_2}, v_{k_2}), \ldots, (v_{i_m}, v_{k_m})\}$ be a finite set of edges. A directed graph is formally, $\mathfrak{G} = (V, E)$, where E and V have just been defined.

In a directed graph diagram, a dot represents a vertex. A line segment with one arrow pointing from one vertex to the other vertex represents a directed edge. The diagram titled, *Directed Graphs* shows one graph that contains a cycle and another that does not contain a cycle so it is called a tree.

A useful analogy is that the structure of hyperlinks on the World Wide Web is a directed graph because it is possible to click on a link to go to web page A, by clicking on a link while at web page B, but not vice versa. In mathematical language, (B, A) is an edge in the directed graph, but (A, B) is NOT an edge.

CGE is the first evolutionary method to evolve directed graphs containing cycles. It is also the first evolutionary method to crossover and mutate these directed graphs containing cycles.

E.) In previous evolutionary methods, each vertex represents a static function. The directed edge represents output from one function delivered as input to the other function. Because of this structure, information does not flow in the opposite direction. Furthermore in a function, the output can not be computed until it receives the input. Consequently, information flows in a synchronized way.

On the other hand, the objects that CGE operates on are entirely different. Effectors have more structure, more flexibility, and more computing capability than functions. Each vertex represents one effector, and each directed edge represents a non-zero connection from effector A to B. Furthermore, it is also possible to have a non-zero connection from effector B to A, so information flows asynchronously in both directions . Most importantly, effectors can change over time, so an effector can mimic the behavior of an infinite number of functions.

F.) CGE can also improve the design of analog VLSI circuits which are suitable for implementing Effector machines in hardware. One of Mead's primary methods of designing subthreshold analog circuits, [MEAD], is to use piecewise linear analysis to design non-linear circuits. Piecewise linear analysis becomes mathematically intractable as the size of the circuit increases. Also, piecewise linear analysis often requires a person to come up with clever techniques for predicting how the circuit will

behave. In other words, it is very difficult to write a computer program that could automate the design of these circuits, using piecewise linear analysis. This greatly increases the time and financial cost of designing these circuits. CGE does not require cleverness. One can treat the analog VLSI circuit as a black box. All you need to know is for a set of inputs $\{I_1, I_2, \dots\}$, what set of corresponding outputs, $\{O_1, O_2, \dots,\}$, are required of the analog VLSI circuit. Then execute CGE on the *(input, output)* set, $\{(I_1, O_1), (I_2, O_2), (I_3, O_3), \dots\}$.

7. HARDWARE IMPLEMENTATION OF EFFECTORS

This section addresses the implementation of an Effector machine in hardware; an example is a semiconductor chip . In a hardware implementation of an Effector machine, computation still can be robustly executed even though there may be a small amount of variance in the transmission time or the exact time an effector is supposed to fire. The analysis below explains how to design an effector machine despite substantial variance in the physical parameters of the hardware.

Let r denote the refractory period of an effector. Let t_{detect} denote the time at which effector E_i detects some other effector fired. Let t_{actual} denote the actual time that this other effector fired. To make the analysis simpler to present, we ignore the transmission time from the effector that fired to the effector that received this firing information. Let ϵ_i denote the maximum possible value for $|t_{detect} - t_{actual}|$. Define $\epsilon = \max\{\epsilon_i : E_i \textit{ is an effector in machine } \mathfrak{M}\}$. If $\epsilon = 0$, then all effectors detect when another effector has fired with perfect precision. Let $\mathbf{T}$ denote a finite interval of time such that the hardware obeys $0 \leq \epsilon < r < \mathbf{T}$. Refer to the diagram, titled *Error Tolerance in Effectors.*

Define $\chi : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ as $\chi(L,n) = L * (L-2) * (L-4) \cdots * (L-2n+2) = \prod_{k=1}^{n} (L-2k+2)$. As an example, $\chi(7,3) = 7 * 5 * 3$. As another example, $\chi(20,5) = 20 * 18 * 16 * 14 * 12$. Choose $\mathbf{T}$ so that r divides $\mathbf{T}$ with zero remainder.

The formula:

$$\sum_{n=1}^{\frac{\mathbf{T}}{r}} \frac{\chi(\frac{\mathbf{T}}{r}, n)(\frac{r}{\epsilon})^n}{n!}$$

is the maximum number of possible distinct firing configurations for an effector, during the interval of time $\mathbf{T}$. This number of firing configurations determines the number of different states that the effector could be in during this time period.

The schematic diagram titled *Effector Circuit* illustrates how one Effector can be implemented with a circuit built from transistors that operate subthreshold. This technique reduces the amount of power consumed by more than 5 orders of magnitude. In Carver Mead's words, for a transistor to operate subthreshold means: "The gate

voltage at which the mobile charge [in the transistor] begins to limit the flow of current is called the threshold voltage .... Most circuits described in this book operate in *subthreshold – their gate voltages are well below the threshold voltage*," [MEAD]. Furthermore, when transistors operate subthreshold, the amount of heat produced is greatly reduced.

VII. BIBLIOGRAPHY

[Bledsoe] Bledsoe, W.W. (1961)
The use of biological concepts in the analytical study of systems.
ORSA-TIMS National Meeting, San Francisco, CA.

[Box] Box, G.E.P. (1957)
Evolutionary operation: A method for increasing industrial production.
Journal of the Royal Statistical Society, C, 6(2), 81-101.

[Bremermann] Bremermann, R.J. (1962)
Optimization through evolution and recombination.
Self-organizing systems. pp. 93-106.
Washington, D.C., Spartan Books.

[Friedman] Friedman, G.J. (1959)
Digital simulation of an evolutionary process.
General Systems Yearbook, 4, 171-184.

[Mead] Mead, Carver. (1989)
Analog VLSI and Neural Systems.
Addison-Wesley Publishing Company. ISBN 0-201-05992-4

[Rall] Rall, Wilfrid. (1995)
The Theoretical Foundation of Dendritic Function.
MIT Press. ISBN 0-262-19356-6.

[Turing] Turing, Alan M. (1936)
On Computable Numbers, with an Application to the Entscheidungsproblem,
Proceedings, London Mathematical Society, 2, no. 42, 230-265, and no. 43, 544-546.